On Formula Embeddings in Neural-Guided SAT Solving[1]

Mert Dumenci

December 2, 2019

[1] Thanks to *Prof. Alessandro Orso*, *David Heath*, *Hanjun Dai* of Georgia Tech College of Computing, *Bill Harris* of Galois, *Metin Say* of MIT, *Mate Soos* of cryptominisat for the discussions and help.

Getting a handle on the consequences of any premises, or at least the fastest method for obtaining these consequences, seems to me to be one of the noblest, if not the ultimate goal of mathematics and logic. Ernst Schröder

Branching heuristics determine the performance of search-based SAT solvers. We note that recently, Neural Machine Learning approaches have been proposed to learn such heuristics from data. The first step in learning a branching heuristic is a transformation from the space of Boolean formulas to a vector space. We note that there is no canonical transformation: techniques such as message-passing networks and LSTMs have been proposed to embed formulas into $\mathbb{R}^n$. We build a novel dataset of an approximate optimal heuristic and compare the estimation performance of models with different embedding methods. We show that for performant models, embedding methods need to represent the structural invariances of Boolean formulas: similar to CNNs and spatially local data such as images.

Introduction

Finding assignments to variables in Boolean formulas such that the formula evaluates to `True` is a problem of utmost theoretical and practical interest. SAT solving, as it's called, defines the formal class of problems in Computer Science where a given solution can be verified quickly, but there are no known ways to quickly *find* such a solution. In practice, SAT is integral in various practical fields such as microprocessor design and machine reasoning. While SAT solving is a heavily researched topic with advanced solvers[2] able to solve formulas with millions of variables in a "reasonable" time, this timescale can reach up to days. Recently, various attempts at phrasing the problem in a Machine Learning setting has shown promising improvements over traditional methods[3]. *We explore a central problem in phrasing SAT solving as connectionist machine learning (ML) problems: how do we represent Boolean formulas as vectors in $\mathbb{R}^n$, and how does this "embedding" of formulas relate to the performance of our task?* We're now going to explore this question in-depth.

[2] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. 2008

[3] Gil Lederman, Markus N. Rabe, Edward A. Lee, and Sanjit A. Seshia. Learning heuristics for automated reasoning through reinforcement learning. 2019

We want to introduce a few fundamental concepts that will be mentioned in this paper ahead of time.

Conjunctive Normal Form (CNF) is a canonical form for any

Boolean formula f, such that f is a conjunction ($\wedge$) of one or more clauses, where a clause is a disjunction ($\vee$) of literals[4].

[4] e.g. $(p \vee q) \wedge z$, or in words, p or q, and z.

Satisfiability Problem (SAT) is the decision problem that given a formula ϕ in CNF, does there exist[5] an assignment to variables, $p = \top$[6], $q = \bot$ and so on, such that ϕ evaluates to $\top$? (More formally, does there exist an *interpretation* $v(\cdot)$ such that $v(\phi) = \top$?)

[5] ϕ is called *satisfiable* if such a v exists.

[6] $\top$ is true and $\bot$ is false.

While this may sound like a trivial problem, it is far from it. The class $\mathcal{NP}$-Complete of problems in computer science was defined by the SAT problem[7], which makes it the de-facto *definition* of a problem that we don't know how to solve fast (but do know how to check a solution fast.) In fact, one of the most important open problems in mathematics[8] is the question $P \stackrel{?}{=} NP$, which can be reduced to the question: is SAT solving fundamentally harder than problems such as determining if a number is prime?

[7] Stephen A. Cook. The complexity of theorem-proving procedures. 1971

[8] Clay Mathematics Institute. Millenium problems, 2019. URL `http://www.claymath.org/millennium-problems`

Perhaps more importantly for our purposes, SAT solving is used frequently in practice. SAT solving is a crucial tool to fields such as automated theorem proving (for proving the correctness of software), circuit design (for designing microprocessors) and symbolic reasoning (checking validity of statements, finding consequences from premises).

While theoretically assumed to be hard, efficient techniques exist for solving SAT problems in practice. It's not our intention to underestimate the state-of-the-art, well-engineered solvers. As we'll explain, the basic ideas and techniques for solving SAT problems are surprisingly intuitive, and are rendered very powerful with the years of heuristic engineering poured into industrial solvers[9].

[9] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. 2008

Conflict-Driven Clause Learning (CDCL) forms the scaffolding for most modern SAT solvers. It's not hard to explain the intuition (fig. 1) behind it. Given a Boolean formula f:

- Pick an undecided variable L according to some function $h(\cdot)$ (which is called a *branching heuristic*).
- Assign it a truth value $T \in \{\bot, \top\}$, evaluate the formula to f'.
- If there's an impossible "conflict" in f' (such as p must be both $\bot$ and $\top$), we know that the solution can't be this. Find the source of the conflict and remember that the solution must *not* repeat this condition[10]. Jump back appropriately in the decision tree, and run again.
- If there are no conflicts and the formula still has undecided variables, run again with f'.

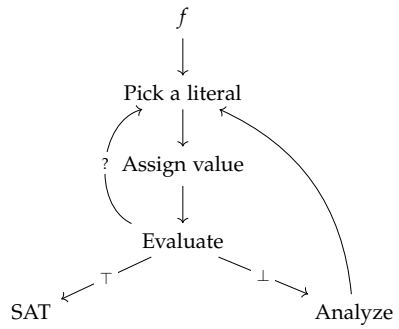


Figure 1: A (simplified) diagram of a Conflict-Driven Clause Learning (CDCL) solver.

[10] This is called clause learning.

- In the end, if we've evaluated f to $\top$, f is SAT. Else, keep running until all assignments are tried.

Let's look at why this problem is hard. For each variable, we have two choices: $\{\bot, \top\}$. We have n variables, thus we can make $\mathcal{O}(2^n)$ decisions and definitively find the result. In practical cases, it's not uncommon to have hundreds of thousands of variables.

$$2^{500000} = 9.95 * 10^{150514}$$

That's a lot of operations. How does this ever work in practice? It's not immediately obvious that $h(\cdot)$ has any effect on the running time of this process. Let's illustrate.

Let $h(\cdot)$ be a heuristic that picks, in-order, q_1 through q_n and p at last, assigning $\top$ first and then $\bot$.

$$f = (p \vee q_1) \wedge (p \vee q_2) \wedge \ldots (p \vee q_n)$$

Observe that the solver will assign $q_1 = \top$ first:

$$\begin{aligned} f &= (p \vee \top) \wedge (p \vee q_2) \wedge \ldots (p \vee q_n) \\ &= \top \wedge (p \vee q_2) \wedge \ldots (p \vee q_n) \\ &= (p \vee q_2) \wedge \ldots (p \vee q_n) \end{aligned}$$

It will then continue assigning $q_i = \top$ until all n literals are set to $\top$, and a satisfying solution is found. Compare this with a better heuristic $h'(\cdot)$ that picks p first. The process will terminate in one iteration: p occurs in every clause, and any assignment with $p = \top$ is a satisfying assignment.

We hope that we can then make the claim efficiently solving SAT problems can be reduced to finding a good heuristic $h^\star(\cdot)$, without sounding all too ridiculous.

Problem

Finding efficient branching heuristics $h^\star(\cdot)$ has been an active area of research ever since work has first been done on a SAT solver. These classical heuristics such as Variable State Independent Decaying Sum (VSIDS) have been engineered by hand, and are proven to be very efficient in practical use cases[11].

A contrasting approach to hand-engineering heuristics is learning them in a machine learning setting. The exponential growth in ML in the recent years has shown that ML-based techniques can be a significant step towards solving previously intractable problems. A natural question one might ask, then, is: can we learn a "good" SAT branching heuristic using ML methods? There are numerous ways to approach this task. For simplicity[12], let's pick supervised learning.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. 2008

[12] We've diverged from our initial aims of building a reinforcement learning environment for learning heuristics due to the computational requirements and fragility of the method.

Supervised learning is estimating a function $f : \mathcal{X} \to \mathcal{Y}$ from a training set $x \times y \subset \mathcal{X} \times \mathcal{Y}$. Assuming we have the training set, we aspire to learn $h(\cdot)$ as f. We finally come to the main topic of our paper: connectionist ML methods (neural networks, etc.) expect vectors as input, but we have Boolean formulas. We, then, have to encode (or, more formally, *embed*) Boolean formulas in a vector space. How do we do this?

We could convert the formula into a string and use any of the well-known text embedding methods. This is a good first step, however, "the semantics of propositional logic induce rich invariances that such a syntactic method would ignore, such as permutation invariance and negation invariance."[13]

[13] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. 2018

In this paper, we aim to answer the question: how critical are the inherent syntactic invariances of Boolean formulas in learning optimal branching dynamics, and thus improving SAT solvers using ML? We build a bounded search tool for improving solver heuristics, and learn an approximation of an optimal heuristic $h'^{\star}$ using different embedding techniques for an empirical study on the effect of formula representation in Neural-Guided SAT Solving.

Existing work/results

Selsam et al.

Selsam et al.[14] exhaustively explores the syntactic invariances of Boolean formulas. They found the satisfiability of a formula is not affected by:

[14] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. 2018

Figure 2: Structural invariances of a Boolean formula

- Permuting the variables (Swap x_1 and x_2.)
- Permuting the clauses (Swap c_1 and c_2.)
- Permuting the literals *within* a clause (Swap $x_2 \vee \neg x_1$ with $\neg x_1 \vee x_2$ in a fixed clause c.)
- Negating every literal corresponding to a given variable. (Replace x_1 with $\neg x_1$ in the formula.)

They proceed to build a message-passing architecture to embed formulas with respect to these invariances, and learn an end-to-end SAT-solver using these embeddings. They do not repeat their experiments with a more traditional embedding.

Mingzhe Wang et al.

Another paper that explores a similar embedding approach is Wang el al.[15] The authors take an ML-based premise selection model and learn vector embeddings of first-order logical formulas *contemporaneously* with its training. Their results improve over the results of the original paper. This is a comparison akin to what we aim to achieve; albeit their results show improvements in premise-election for theorem-proving and not our domain.

[15] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. 2017

Fei Wang et al.

Wang et al.[16] (not the same authors as the previous paper) explore learning branching heuristics in a reinforcement learning setting similar to AlphaGo Zero[17]. This is a reinforcement learning approach to what we're proposing to do, except the authors do not use a graph embedding for the formulas; thus not representing the syntactic invariances. It's important to note that all of their converged models outperformed a state-of-the-art SAT solver. They do not repeat their experiments with a more complex embedding that keeps syntactic invariances intact.

[16] Fei Wang and Tiark Rompf. From gameplay to symbolic reasoning: Learning SAT solver heuristics in the style of AlphaGo Zero. 2018

[17] David Silver et al. Mastering the game of Go with deep neural networks and tree search. 2016

Lederman et al.

Lederman et al.[18] explore learning branching heuristics for solving Quantified Boolean Formulas (QBF) using a graph neural network (GNN) embedder, followed by a policy network for predictions. Their method is similar to the Selsam et al. method, but they do not repeat their experiments on a simpler embedding.

They note their reasoning for not using a recurrent network architecture for embedding formulas:

> Most previous approaches that applied neural networks to logical formulas used LSTMs or followed the syntax-tree. We believe that this approach is inherently limited as variables can occur in distant parts of a formula and modeling formulas as sequences or as trees therefore requires networks to remember the complete formula, which is impractical beyond small formulas. This intuition seems to be confirmed by the fact that neither of these approaches scales to formulas of significant sizes.

[18] Gil Lederman, Markus N. Rabe, Edward A. Lee, and Sanjit A. Seshia. Learning heuristics for automated reasoning through reinforcement learning. 2019

Methods

Dataset

To the best of our knowledge, no dataset exists for a *perfect CDCL heuristic* $h^\star(\cdot)$. The worst case time complexity of generating such a heuristic is $\mathcal{O}((2n)!)$, as the heuristic must try every assignment to every variable in every order and pick the best:

$$\begin{aligned}
&(p = \bot),\ (q = \top),\ \dots \\
&(p = \top),\ (q = \bot),\ \dots \\
&(q = \top),\ (p = \bot),\ \dots \\
&(q = \bot),\ (p = \top),\ \dots \\
&\qquad\qquad\qquad\qquad \dots
\end{aligned}$$

Since the search space of formulas with a small number of (< 20) variables are small (≤ 1 million) any *interesting* optimal heuristic will have to come from larger formulas. Then, on a 20 variable formula, finding the perfect heuristic will need

$$\begin{aligned}
(20 * 2)! &= 40! \\
&\approx 8 * 10^{47} \text{ steps}
\end{aligned}$$

This is infeasible.

To find an approximation for the optimal heuristic, we've built a tool that incrementally improves an existing SAT solver's heuristic[19]. We're interested in improving its heuristic, as we don't want to simply learn the SAT solver's internal, hand-coded heuristic. Our goal is to learn a function that has more *insight* than an human engineered one.

[19] We're using cryptominisat as the base solver.

Given a formula, we run a bounded search[20] on the problem's search space and assign all available variables to all truth values, running the SAT solver on the resulting formulas. The SAT solver responds with how many decisions it made to solve the resulting formula. We pick the branch that reported the minimum number of remaining decisions, and call it $h'^\star$.

[20] Currently 1 level.

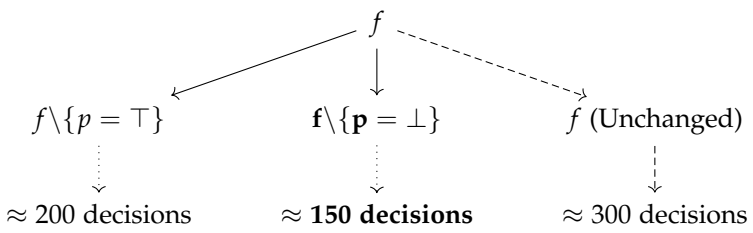


Figure 3: Estimating $h^\star(\cdot)$ with a bounded search

Lemma 1. $h'^{\star}$ can *only be better* than the SAT solver's internal, opaque heuristic.

Proof. The SAT solver's heuristic *must* take a branch. Call this branch b. $h'^{\star}(\cdot)$ takes *all* possible branches, including b, and picks b' with the fewest number of decisions. If $b = b'$, then $h'^{\star}(\cdot)$ is equal to the SAT solver's heuristic. If not, b' is a valid heuristic that makes a fewer number of decisions, and $h'^{\star}(\cdot)$ is better than the solver heuristic. □

For our formulas, we sample from Selsam et al.[21]'s

$$\mathcal{SR}(20)$$

distribution of randomly generated SAT formulas. There are 20 variables in these formulas, and clause lengths sampled from

$$1 + \text{Bernoulli}(0.7) + \text{Geo}(0.4)$$

All of our formulas are *satisfiable* for simplicity: our main focus is to explore encodings for formulas to best learn a heuristic, and not build an end-to-end state of the art SAT solver. We expect our findings to generalize to an ML system that can handle unsatisfiable formulas. For normalization, we rename all formulas in our datasets: any formula that our ML models see have their variables renamed using the same procedure[22].

Empirically, $h'^{\star}(\cdot)$ on our test set of formulas[23] resulted in an $\approx 2.4\times$ reduction in the number of decisions over the solver heuristic.

[21] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. 2018

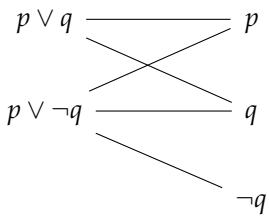
[22] Starting from 1 and reading the formula left-to-right, the formulas are renamed incrementally.

[23] 606136 formulas sampled from $\mathcal{SR}(20)$.

Predictor

We build two multi-layer neural networks for estimating $h'^{\star}(\cdot)$:

Option A: Graph Embedding of Literals $G(d, T)$ We implement a modified version of the Selsam et al.[24] method in `pytorch`[25].



In this method, formulas are represented as a bipartite graph with clauses and literals as nodes, and edges between two nodes *if and only if* a literal is contained within a clause. Note that this representation erases all positional information from literals: any ordering of clauses and literals within clauses will give the same

[24] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. 2018

[25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017

resulting graph. This encoding satisfies the formula invariances listed by Selsam et al. (fig. 2)

A *message passing* model $G(d, T)$ is run on this graph. Starting from initial clause and literal embeddings C_{init} and L_{init}, at each iteration t:

- Every clause embedding C_i is updated:

$$L_{agg} = \sum_{L_i \in N(C_i)} L_{msg}(L_i^t)$$
$$C_i^{t+1} \leftarrow C_u(C_{h,i}^t, L_{agg})$$

  Where C_u is a three-layer LSTM with hidden size d, C_{uh} is the hidden state of C_u, L_{msg} is a three-layer feed-forward network with all layers of size d and $N(C_i)$ is the set of literals in the neighbourhood of C_i.

- Similarly, every literal embedding L_i is updated:

$$C_{agg} = \sum_{C_i \in N(L_i)} C_{msg}(C_i^t)$$
$$L_i^{t+1} \leftarrow L_u(L_{h,i}^t,\ \text{concat}(C_{agg}, L_{\neg i}^t))$$

  Where L_u is a three-layer LSTM with hidden size d, L_{uh} is the hidden state of C_u, C_{msg} is a three-layer feed-forward network with all layers of size d, $N(L_i)$ is the set of clauses in the neighbourhood of L_i and $L_{\neg i}$ is the literal embedding for the negated version of literal i in the graph.

We run $G(50, 25)$ after which it generates embedding vectors for literals and clauses.

We apply L_{score} on every literal embedding and get *scores* of literals in $\mathbb{R}^n$, where L_{score}[26] is a three-layer feed-forward network of hidden layers of size d, and output layer of size 1. A *softmax* of these scores is returned as the final output of the network[27].

If the network predicts a positive polarity literal l, this is interpreted as branching on $l = \top$. If the network predicts a negative polarity literal $\neg l$, this is interpreted as branching on $l = \bot$.

[26] Selsam et al. calls a similar ranking network L_{vote} in their paper; but our literals are not *voting* for a binary result like in their task.

[27] We turn Selsam et al.'s binary classification problem into a multiclass classification problem. This is the main difference between our graph embedding model and NeuroSAT.

Option B: LSTM Embedding of Formula $L_f(d)$ We encode the formula as a sequence of tokens, where each literal and connective are encoded in one-hot form. In a formula with n variables, each token is in $\mathbb{R}^{2n+2}$ where the first n elements of the vector is for positive polarity literals, second n for negative polarity, and the last two indices for the connectives $\wedge$ and $\vee$.

A uni-directional three-layer LSTM of hidden size d is run on the formula, resulting in an *embedding* f_{embed} of the formula in $\mathbb{R}^d$.

f_{embed} is passed through a three-layer feed-forward network of hidden layer sizes d, and output size $2n$ similar to *Option A.*

Note that this method writes the formula down in a linear form as a human would do, and encodes the entire formula into a single vector in $\mathbb{R}^d$. This method does not respect any of the invariances listed by Selsam et al. (fig. 2) Most importantly, positions of literals are considered in the input to the model. Note that positions of literals in f do not carry any meaning: any permutation of them results in the *same* formula. We run $L_f(50)$.

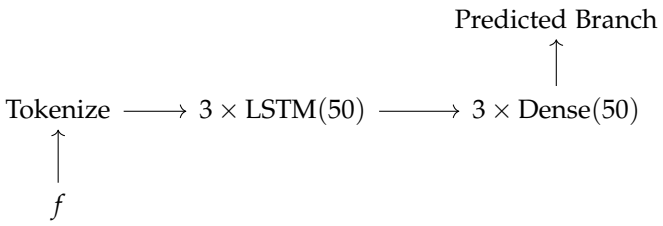


Figure 4: Topology of $L_f(50)$

Option C: LSTM Embedding of Literals $L_l(d)$ We start with the architecture detailed in *Option B*. Instead of running the entire formula through a uni-directional LSTM, we run it through a bi-directional LSTM and extract the last-layer hidden state $f_{embed,i}$ of the LSTM at each literal i.

Observe that by the bi-directional property of the LSTM, for each literal i we are effectively running an LSTM from the first literal to i and backwards from the last literal to i, concatenating the resulting hidden states to $f_{embed,i}$. Then our literal embeddings are in $\mathbb{R}^{2d}$.

We pass these embeddings through L_{score} defined in *Option A*, and similarly *softmax* the results. Note that similar to *Option B*, this method does not respect the invariances listed by Selsam et al. (fig. 2) We run $L_l(50)$.

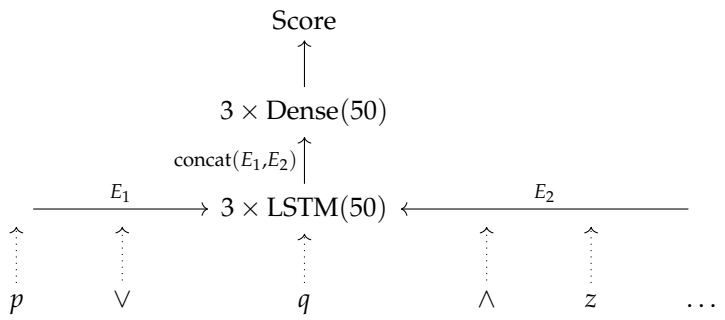


Figure 5: Topology of $L_l(50)$

Training

We train[28] the described networks on the our dataset ($\mathcal{SR}(20)$, 606136 data points) using `pytorch` in a 90% train, 10% test split.

[28] All of our code is open source at https://github.com/mertdumenci/sat-ml.

$G(50, 25)$ The graph encoding model is trained on a single *NVIDIA Tesla V100* GPU for 12 epochs which took approximately 10 hours.

$L_f(50)$ The formula encoding LSTM model is trained on 4 *NVIDIA GeForce RTX 2080 Ti*s for 30 epochs which took approximately 20 hours.

$L_l(50)$ The literal encoding LSTM model is trained on 4 *NVIDIA GeForce RTX 2080 Ti*s for 5 epochs which took approximately 14 hours[29].

[29] This model was $\approx 8\times$ harder to train than $G(50, 25)$.

Results on $\mathcal{SR}(20)$

It should be noted that classes in $\mathcal{SR}(20)$ are approximately *uniformly* distributed. A naive predictor that *does not* use any information in the formulas will be as successful as chance, which is $\frac{1}{40} = 2.5\%$ in accuracy terms. We set this as our baseline.

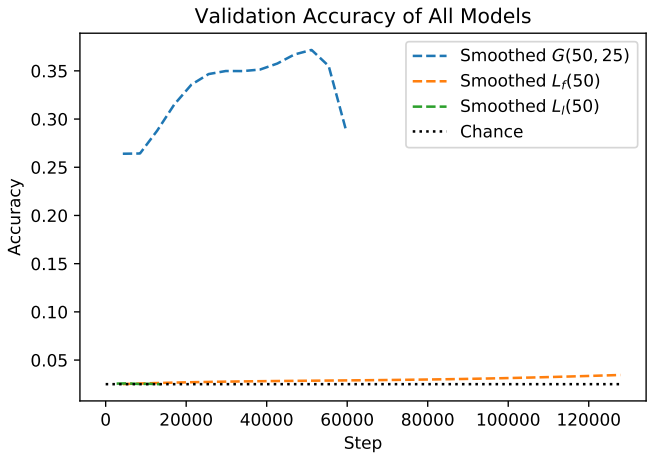


Figure 6: All models performance on $\mathcal{SR}(20)$

$G(50, 25)$

The graph embedding model (fig. 7) with 25 iterations and an embedding dimension of 50 quickly reaches a validation accuracy of $\approx 26\%$ in the first epoch of training. This is $10\times$ more accurate than our baseline of chance. The validation accuracy and loss curves increases steeply with more epochs, reaching a maximum of 36%.

Validation accuracy takes a steep dive after 12 epochs–we attribute this to the model overfitting due to the training accuracy continuing

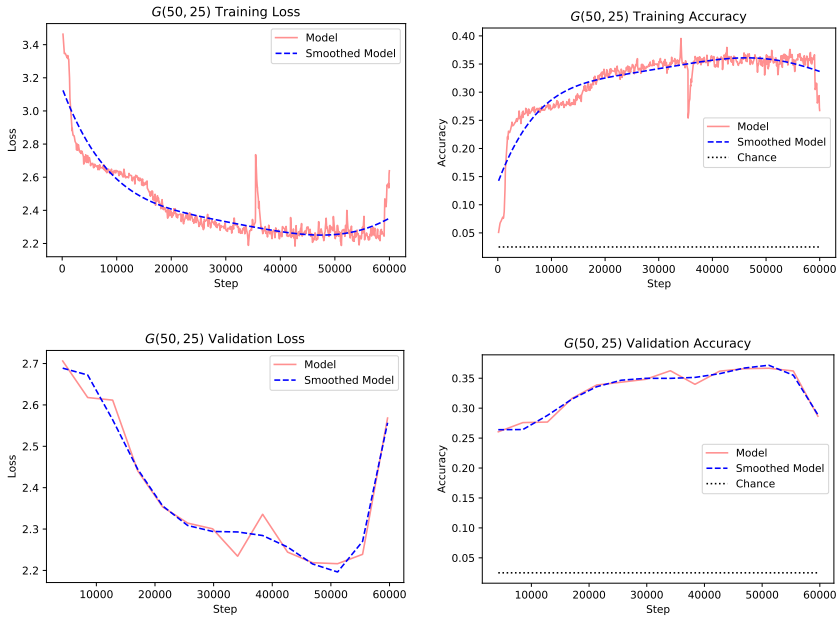


Figure 7: $G(50, 25)$ performance on $\mathcal{SR}(20)$

its rise. We predict that a better performance can be achieved with a higher dimensional model ($d > 25$) and more data, both of which can be achieved with larger amounts of compute.

$L_f(50)$

The formula-embedding LSTM model (fig. 8) with a hidden dimension of 50 reaches a maximum validation accuracy of 3.5% throughout the 30-epoch training run. While the validation accuracy curve shows an upwards curve, the rate of increase is negligible when compared to $G(50, 25)$ (fig. 6). We end training at 30 epochs due to the slow rate of improvement.

$L_l(50)$

The literal-embedding LSTM model (fig. 9) with a hidden dimension of 50 reaches a maximum validation accuracy of 2.6%. The model immediately overfits to the dataset, and the validation accuracy curve takes a dive after the first epoch. Training curves stabilize, and the model shows no premise for improvement. We end training at 5 epochs due to these reasons.

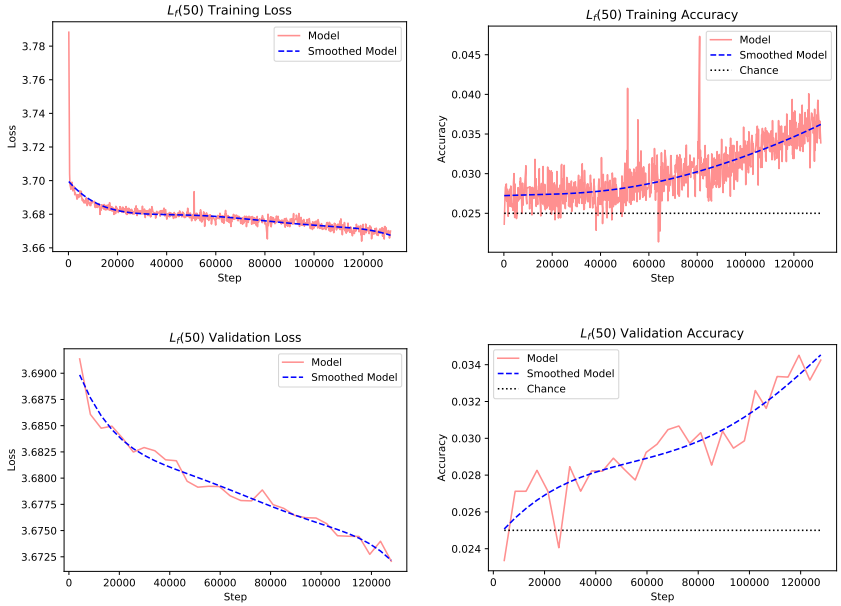


Figure 8: $L_f(50)$ performance on $\mathcal{SR}(20)$

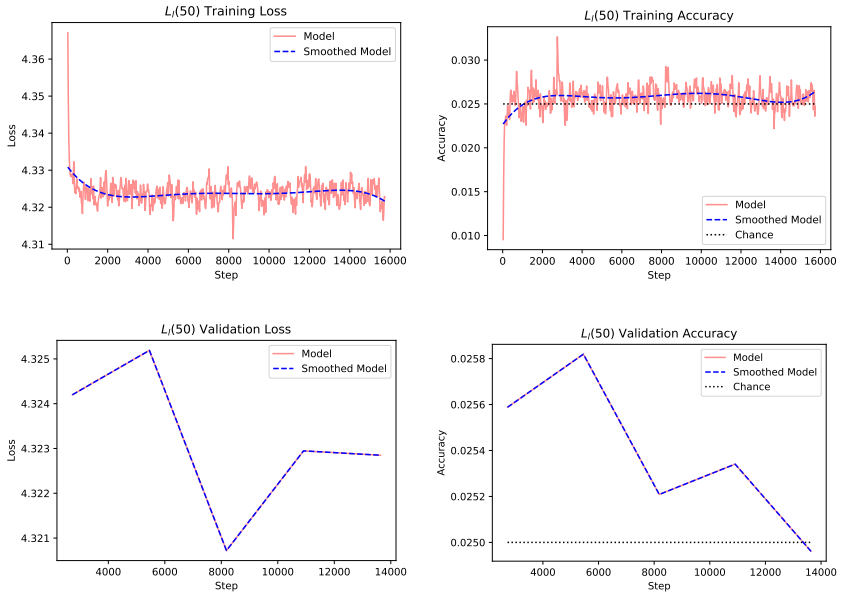


Figure 9: $L_l(50)$ performance on $\mathcal{SR}(20)$

Model	Avg. acc.	Over chance	Ratio to over chance on $\mathcal{SR}(20)$	Ratio to accuracy on $\mathcal{SR}(20)$
$G(50, 25)$	15%	12×	0.82×	0.41×
$G(50, 250)$	28%	22.4×	1.53×	0.77×
$L_f(50)$	1.43%	1.12×	0.82×	0.40×
$L_l(50)$	1.29%	1.04×	1.00×	0.50×

Table 1: Evaluating $\mathcal{SR}(20)$ model performance on $\mathcal{SR}(40)$

Results on $\mathcal{SR}(40)$

In this experiment, we test our models' generalization performance to a higher number of variables. We take our models trained on $\mathcal{SR}(20)$ and fix their maximum number of variables[30] to 40. The models have *never seen* more than 20 variables in training, but they support such formulas. We assess their performance on a validation set of 10000 formulas in $\mathcal{SR}(40)$.

[30] The graph model is unchanged due to its architectural invariance to the number of variables.

Note that changing the iteration parameter t on our graph model *does not* require retraining. We can train the model on any t_0, and run on any t–each iteration is simply a repeated application of the message passing equations described in *Predictor*. In addition to testing $G(50, 25)$'s generalization performance, we also run it for 250 iterations and note its results[31]. See table 1.

[31] Selsam et al. note in the original NeuroSAT paper that generalizing SAT solving to a higher number of variables require more iterations.

Discussion

To summarize the results, the graph embedding method $G(50, 25)$ shows a stellar performance on the task of estimating an ideal branching heuristic at 10× better than chance, while the LSTM based methods $L_f(50)$ and $L_l(50)$ do not perform significantly better than chance. Further, we can argue that $G(50, 25)$ shows ample opportunity of improvement with a larger dataset and a higher embedding dimension, while $L_f(50)$ and $L_l(50)$ are unlikely to match the graph model in performance in any comparable amount of time, number of model parameters or data due to their demonstrated slow improvement.

In our study of generalization, we see that $G(50, 25)$ loses approximately half its accuracy when generalizing to formulas with 2× the number of variables, but only loses 18% of its accuracy when compared to chance on both inputs. We could argue that either is a better metric than other: ratio over chance for model evaluation, since a higher number of variables means a higher-dimensional classification problem; and accuracy for practical reasons, since we'd want the heuristic to behave accurately regardless of the dimensionality of the input. Without changing the model, we also run it for 250 iterations ($G(50, 250)$) on $\mathcal{SR}(40)$. The accuracy of the model doubles

to 28%, and the model behaves *better* than the original model when compared to chance. This is a staggering result: the model learns an iterative process that predicts the optimal heuristic, which can be run for a higher number of iterations on higher-dimensional input *without changing the learned model*. This is the same conclusion Selsam et al.[32] find in applying the graph model on the problem of solving SAT: they predict that the model learns a search algorithm that mirrors conventional SAT solving, which can be run for any number of iterations until convergence. The LSTM-based models perform similarly on 40 variables as they do with 20–they perform roughly equal to chance.

[32] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. 2018

Embedding methods representing Boolean formulas' structure and invariances perform significantly better than methods that don't. Our graph based model performed an order of magnitude better than chance, while the LSTM-based models have performed roughly equal to chance. The performance of the models suggest that $G(50, 25)$ learned an inherent property about formulas and branch prediction that it could generalize to higher dimensional inputs, while $L_*(50)$ performed negligibly better than chance by learning superficial heuristics. We've noted before that treating a logical formula as a string of characters flattens the structure of the formula and erases all invariances that Selsam et al. list (fig. 2) We've demonstrated that representing such invariances in the model's inductive bias is critical for learning properties about Boolean formulas, which the predictor network uses to predict branches.

We can attribute $G(50, 25)$'s performance to a weak duality between estimating $h'^{\star}$ and SAT solving. By construction, the perfect heuristic we're estimating always picks a branch such that its pick $p \in \{\top, \bot\}$ is in the set of correct interpretations for formula f. This is implied by our selection of the branch with the lowest number of remaining decisions: if a branch is *not contained* in a satisfiable assignment, the solver has to backtrack and take another branch. This enforces that an unsatisfiable branch will always have more remaining decisions than a satisfiable branch, and we pick the minimum of all branches. Then we can argue that if we correctly learn the optimal heuristic using any of our models, and that model generalizes to formulas of any size perfectly, we have solved SAT. We can follow the heuristic, and we'll find a satisfiable solution to f in linear time. The initial conception of the graph model in Selsam et al.[33] demonstrates that an architecture almost identical to $G(50, 25)$ can solve SAT on $\mathcal{SR}(\mathcal{U}(10, 40))$ with 85% accuracy. While our tasks are different, the complexity of them are similar due to the weak duality–to which we

[33] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. 2018

attribute $G(50, 25)$'s performance. This finding is further corroborated by the demonstrated iterative nature of the graph model, as the original NeuroSAT behaves similarly.

The graph model generalizes well to higher dimensional input due to the iterative nature of the learned problem. An important factor for the practicality of our models is their generalization performance to more variables. We should note that the graph embedding model does not have the number of variables n as an input to the model: trained on n variables, it can run on any m variables without modification. This is made possible by its structural representation of a Boolean formula: names of variables are disregarded in the graph representation, and information is extracted based on local structural properties such as variable-clause containment. Contrary to this, the LSTM-based models take variable names as input which requires the model to learn to unlearn this information[34] and restrict the maximum number of variables to a fixed n due to the one-hot encoding of literals. Further, our experiments in generalization (table 1) have shown that while accuracy is reduced if the graph model is run on higher dimensional input with the same number of iterations, the same model performs comparable to its original input size when run for a higher number of iterations. This suggests that the graph model has learned an *iterative process* that generalizes to higher dimensions with a higher number of iterations, without the need for retraining the model.

[34] The graph model disregards names as an inductive bias, while the LSTM models have to learn to disregard.

Based on the results of $G(50, 25)$, we believe that a similar model can be a practical competitor to hand-written heuristics such as VSIDS[35]. Trained on higher dimensional embeddings and a larger dataset beyond our means of compute, we believe that $G(\cdot, \cdot)$ can more accurately estimate an optimal heuristic. Our results need not be perfect: search-based solvers (such as CDCL) accommodate mistakes. In the case of a misprediction, search-based solvers can backtrack and continue searching for an answer.

[35] After optimizations on inference performance. A hand-written heuristic is fast to execute, while $G(50, 25)$ is not.

Conclusion

We define the optimal branching heuristic $h^\star$ as the branching strategy that guides a search-based SAT solver to a satisfying assignment in the lowest number of decisions. We estimate this heuristic ($h'^\star(\cdot)$) for formulas of size n using a bounded search, pruning the search tree using a SAT solver (fig. 3.) We define three models to learn this heuristic: a graph-based model $G(50, 25)$ that respects formula structure and invariances, and LSTM-based models $L_f(50)$ and $L_l(50)$ that linearize the formula and learn on a linear stream of tokens. All three models use similar predictor networks after the embedding steps.

We find that the graph-based model performs an order of magnitude better than chance, and generalizes to a higher number of variables without a steep decline in accuracy. The LSTM-based models do not perform notably better than chance in their training number of variables or in higher dimensions. We argue that these results indicate the graph model, with its representation of formula structure in its inductive bias, learns inherent abstract properties about the formulas that the predictor network uses to estimate $h'^{\star}(\cdot)$. The flattening of formula structure in the LSTM-based models erases this information, and the models fail to converge to a state that accurately predicts optimal branches. We describe a weak duality between estimating an optimal branching heuristic and estimating satisfiability (SAT solving), and use this duality to explain the performance of the graph model. We conclude by arguing that the accurate representation of Boolean formula structure and invariances are essential in any machine learning model that works in the Boolean formula domain.

Future Work

Future questions to address include how to improve the graph model $G(\cdot, \cdot)$ with a larger dataset and embedding dimension, how it compares to a traditional heuristic such as VSIDS when integrated into a CDCL solver, and a more exhaustive analysis of generalization with higher dimensional test inputs. Particularly, with larger compute, we're interested in exploring the asymptotic growth in the number of iterations needed to maintain accuracy on higher dimensional inputs.

References

Stephen A. Cook. The complexity of theorem-proving procedures. 1971.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. 2008.

David Silver et al. Mastering the game of Go with deep neural networks and tree search. 2016.

Clay Mathematics Institute. Millenium problems, 2019. URL `http://www.claymath.org/millennium-problems`.

Gil Lederman, Markus N. Rabe, Edward A. Lee, and Sanjit A. Seshia. Learning heuristics for automated reasoning through reinforcement learning. 2019.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. 2018.

Fei Wang and Tiark Rompf. From gameplay to symbolic reasoning: Learning SAT solver heuristics in the style of AlphaGo Zero. 2018.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. 2017.